Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Системное программирование

Когутич Денис Александрович

Реализация механизма поддержки ограничений в проекте WMP

Выпускная квалификационная работа

Научный руководитель:
к.т.н. Литвинов Ю. В.

Рецензент:
аспирант каф. системного программирования Перешеина А. О.

Санкт-Петербург
2017

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems
Software Engineering

Denis Kogutich

Implementation of mechanism for supporting constraints in WMP project

Graduation Project

Scientific supervisor:
Candidate of Engineering Sciences Litvinov Y. V.

Reviewer:
Ph.D. student Peresheina A. O.

Saint-Petersburg
2017

Оглавление

Введение

Языки графического моделирования являются предпочтительным выбором при проектировании разрабатываемой системы, когда дело доходит до определения структурных аспектов, а именно: основных понятий, свойств и отношений. Наиболее популярным примером данной графической нотации является UML.

Однако UML-диаграммы, как правило, не предоставляют достаточных инструментов для того, чтобы обеспечить и описать все соответствующие аспекты спецификации. Существует, кроме всего прочего, необходимость описывать дополнительные ограничения на объекты модели. Такие неструктурные ограничения часто описываются на естественном языке. Практика показывает, что это всегда приводит к неоднозначностям. Для описания данных недвусмысленных ограничений были разработаны так называемые формальные языки. Недостаток традиционных формальных языков в том, что они могут использоваться людьми с сильной математической подготовкой, но сложны для остальных.

Для разрешения перечисленных выше проблем был создан язык задания ограничений Object Constraint Language (OCL). Это формальный язык для описания не структурных ограничений, а условий, накладываемых на состояние системы [5].

OCL-выражения гарантированно не несут за собой «скрытых эффектов», то есть в процессе вычисления конструкций на данном языке объекты модели не изменяются. OCL используется в первую очередь для проверки конкретного состояния системы на согласованность. Например, с помощью диаграмм классов UML мы не можем указать какие значения может принимать конкретное поле класса и как значение данного поля вычисляется. В случае с OCL достаточно написать набор ограничений и в процессе работы системы проверять их выполнение, если хоть одно из ограничений нарушено – система находится в несогласованном (ошибочном) состоянии. Использование данной информации остаётся на откуп программистам, можно выдать сообщение об ошибке,

либо в случае серьёзных нарушений прервать работу системы.

Также OCL-ограничения могут использоваться в качестве дополнительной документации к системе. Более того, в нынешних реалиях программирования часто используется смесь языков, на которых ведётся разработка, поэтому описание ограничений на специализированном для этого языке является очевидным преимуществом. Нет необходимости прятать проверку ограничений глубоко в коде, когда есть возможность хранить их в специально отведённом месте.

На кафедре системного программирования СПбГУ разрабатывается проект WMP (Web modeling project) [9] – онлайн-платформа предметно-ориентированного моделирования, представляющая собой объединение редакторов диаграмм различных языков, она имеет возможность связи с роботом и возможность отправлять ему диаграммы на исполнение, а так же функционал 2D модели [11]. Также в ней идёт разработка поддержки метаредактора и стандарта BPMN.

Хотелось бы иметь возможность задавать ограничения в проекте WMP, например, для диаграмм поведения роботов можно ввести ограничения следующего вида: после блока «Моторы вперёд» не может сразу идти блок «Моторы стоп», на диаграмме обязан быть только один блок «Начало» и т.д. Это были всего лишь одни из немногих примеров возможных ограничений на состояние системы, в действительности их может быть огромное количество.

1. Постановка задачи

Целью данной квалификационной работы является реализация механизма задания ограничений в проекте WMP. Она заключается в поддержке таких подножеств языка OCL как The Essential OCL и The Complete OCL в данной среде для того, чтобы можно было задавать ограничения на специализированном для этого языке, являющимся мировым стандартом в этой области.

Для достижения этой цели выделены следующие задачи:

- проанализировать существующие OCL-решения;
- реализовать парсер The Essential OCL и The Complete OCL;
- реализовать интерпретатор OCL-выражений из The Essential OCL и The Complete OCL;
- внедрить поддержку OCL в проект WMP.

2. Обзор

2.1. Обзор выбранного подмножества OCL

В данной работе сделан упор на реализацию такого подмножества OCL, как The Essential OCL и The Complete OCL.

The Essential OCL – основная функциональность OCL, поддерживающая выражения над моделями. Сама по себе она практически не используется, так как отсутствует способ передачи моделей, над которыми должны вычисляться выражения. Иными словами, The Essential OCL включает в себя базовые вещи, например, выражения if-then-else-endif, выражения let и так далее.

The Complete OCL предоставляет способ указания существующих моделей, над которыми будут вычисляться OCL-выражения, благодаря ему появляются конструкции context, inv, def, pre, post и так далее.

Цель данного раздела – провести краткий обзор выбранного подмножества OCL и его возможностей. За полной информацией следует обращаться к спецификации [5], либо к документации OCL от создателей Eclipse OCL [3].

OCL является декларативным языком, выражения на нём гарантированно не несут за собой скрытых эффектов [7]. Когда OCL-выражение вычисляется, оно просто возвращает значение, не изменяя состояние модели, на которой оно выполняется.

OCL – типизированный язык, помимо привычных для программистов типов, таких как String, Integer,..., существуют и необычные: OclVoid, OclInvalid, OclAny (базовый тип для всех остальных).

2.1.1. Ключевое слово self

Каждое OCL-выражение находится в контексте экземпляра некоторого специфического типа. Ключевое слово **self** используется для обращения к экземпляру контекста.

Рассмотрим пример из спецификации [5]:

```
context Company
inv: self.numberOfEmployees > 50
```

За ключевым словом **inv** следует объявление инварианта, если в какой-то из моментов выполнения программы он не будет выполняться, значит ограничение нарушено и состояние системы недопустимо. В данном выражении ключевое слово **context** используется для указания контекста, это означает, что описанный ниже инвариант будет проверяться для всех экземпляров класса Company. В самом условии мы проверяем, что у всех экземпляров Company значение поля numberOfEmployees больше 50.

Также существует синтаксис, позволяющий добавить псевдоним для **self**.

Следующий пример эквивалентен предыдущему:

```
context c : Company
inv: c.numberOfEmployees > 50
```

2.1.2. Выражения inv

Под инвариантом класса в OCL понимается условие, которому должны удовлетворять все объекты данного класса. Если говорить более точно, инвариант класса – это логическое выражение, вычисление которого должно давать true при создании любого объекта данного класса и сохранять истинное значение в течение всего времени существования этого объекта [10]. Пример инварианта был продемонстрирован выше. Здесь хотелось бы отметить, что можно указать имя инварианта и несколько инвариантов для одного контекста:

```
context Company
inv: self.manager.isUnemployed = false
inv invName: self.numberOfEmployees > 50
```

2.1.3. Выражения pre и post

OCL-выражения могут быть частью предусловий и постусловий, связанных с некой операцией модели. Предусловия и постусловия следуют за ключевыми словами **pre** и **post**, соответственно.

Пример:

```
context Typename::operationName(param1 : Type1, ... ): ReturnType
pre : param1 > ...
post: result = ...
```

Также **pre** и **post** имеют опциональные имена, по аналогии с **inv**. В постусловии есть доступ к результату вызова операции (**result**).

В **post** выражениях можно получить доступ к значению переменной до вызова операции с помощью добавления "@pre" к имени переменной.

Пример:

```
context Person::birthdayHappens()
post: age = age@pre + 1
```

2.1.4. Выражения body

OCL-выражения могут использоваться для отображения того, что происходит при вызове операции модели с помощью **body**.

Пример:

```
context Person::getCurrentSpouse() : Person
body: self.mariages->select( m | m.ended = false ).spouse
```

2.1.5. Выражения init и der

OCL-выражения могут использоваться для отображения начального (init) и постоянного (derive) значений переменной модели. Пример:

```
context Typename::attributeName: Type
init: -- some expression representing the initial value
context Typename::assocRoleName: Type
derive: -- some expression representing the derivation rule
```

Ограничение **derive** включает в себя **init** и должно быть выполнено в любой момент.

2.1.6. Базовые типы

Базовые типы OCL представлены в таблице 1, можно отметить, что они практически повторяют типы в обычных языках программирования. Помимо данных примитивных типов существуют 4 вида коллекций (Set, Bag, Sequence, OrderedSet) и тип Tuple, позволящий объявить объект. Коллекции отличаются наборами операции, также Set и OrderedSet хранят уникальные значения. OrderedSet и Sequence являются упорядоченными коллекциями, но в действительности сортировка значений не производится, разница заключается в наборах операций и реализациях операций. Также хочется отметить, что коллекция, содержащая значение invalid, согласно спецификации вычисляется как invalid. Но в то же время коллекции могут иметь значения null.

type	**values**
OclInvalid	invalid
OclVoid	null, invalid
Boolean	true, false
Integer	1, -5, 2, 34, 26524, ...
Real	1.5, 3.14, ...
String	'To be or not to be...'
UnlimitedNatural	0, 1, 2, 42, ..., *

Таблица 1: Базовые типы с примерами.

2.1.7. Выражения let

Выражения **let** позволяют объявить одну или несколько переменных для использования в выражении. Переменная, объявленная в **let** должна иметь указанный тип и начальное значение.

Пример:

```
context Person inv:
let income : Integer = self.job.salary->sum() in
    if isUnemployed then
        income < 100
    else
        income >= 100
    endif
```

2.1.8. Выражения def

Выражения **let** позволяют объявить переменную для использования в выражении, следующим за ключевым словом **in**, однако объявленную таким способом переменную нельзя переиспользовать в остальных выражениях. Более того, нет возможности объявить функцию. Данные проблемы решает **def**.

Пример:

```
context Person
def: income : Integer = self.job.salary->sum()
def: nickname : String = 'Red'
def: hasTitle(t : String) : Boolean = self.job->exists(title = t)
```

Область видимости объявлений с помощью **def** – текущий контекст.

2.1.9. Инфиксные и префиксные операторы

Операторы $+$, $-$, $*$, $/$, $=$, $<>$, $<$, $>$, $<=$, $>=$ используются в качестве инфиксных.

Следует отметить, что в OCL выражения **a** $+$ **b** и **a**.$\_'+'$(**b**) эквивалентны, аналогично для остальных инфиксных операторов.

Операторы $-$, **not** используются в качестве префиксных.

В случае с префиксными операторами, эквивалентными являются, например, выражения **-1** и **1**.$\_'-'$().

2.1.10. Навигационные операторы

В OCL существуют два вида навигационных операторов: "." и "->". Оператор "." используется для навигации от объекта по свойству или операции.

Пример:

```
anObject.name
aString.indexOf(':')
```

Оператор "->" используется для навигации от коллекции по свойству, операции или итерации.

Пример:

```
aBag->elementType
aSet->union(anotherSet)
aSet->collect(name)
```

Кроме того, существует синтаксический сахар для навигаций. При навигации от коллекции с использованием "." происходит преобразование следующего рода:

```
aSet.name  преобразуется в  aSet->collect(name)
```

При навигации от объекта через "->" происходит неявное преобразование к коллекции Set.

```
anObject->union(aSet) преобразуется в anObject.oclAsSet()->union(
    aSet)
```

2.1.11. Комментарии

OCL поддерживает два вида комментариев: многострочные и однострочные. Однострочные начинаются с --, пример: --comment. Многострочные комментарии начинаются слешем-звездочкой «/*» и заканчиваются звездочкой-слэшем «*/».

2.2. Обзор существующих реализаций OCL

В данной квалификационной работе было принято решение разбирать OCL-выражения на стороне клиента, соответственно реализовывать парсер и интерпретатор OCL-выражений предстоит на Javascript/Typescript. К сожалению, на данный момент не существует достойных open-source реализаций OCL на Javascript, одна из немногих существующих реализаций – OCL.js [6]. Однако, существуют хорошие реализации на других языках, например Eclipse OCL, который написан на Java. Обратившись к реализации, предлагаемой Eclipse [4], можно найти много интересного.

В первую очередь, Eclipse OCL используют (по крайней мере использовали раньше) генератор парсеров ANTLR Tool [1], который генерирует парсер по описанию EBNF-грамматики. Более того, Eclipse OCL используют обобщенную грамматику OCL [3], отличающуюся от описанной в спецификации.

2.3. Неоднозначность грамматики OCL

Грамматика OCL 2.4, описанная в спецификации [5], является неоднозначной. В связи с этим для многих синтаксических правил существуют уточнения, так называемые правила разрешения неоднозначностей.

Пример из спецификации:

```
LetExpCS ::= 'let' VariableDeclarationCS LetExpSubCS

Disambiguating rules
[1] The variable name must be unique in the current scope.
[2] A variable declaration inside a let must have a declared type
    and an initial value.
```

2.4. VOCL

Помимо текстовых языков задания ограничений, существуют и визуальные языки, специализирующиеся на данной проблеме. Визуаль-

ный «брат» языка OCL – это VOCL (Visual Object Constraint Language) [2], пример ограничения на VOCL продемонстрирован на рис. 1. Было принято решение использовать в работе именно OCL из-за громоздкости и неочевидности VOCL, тем более парсер OCL можно впоследствии переиспользовать в проектах, не связанных с визуальным моделированием.

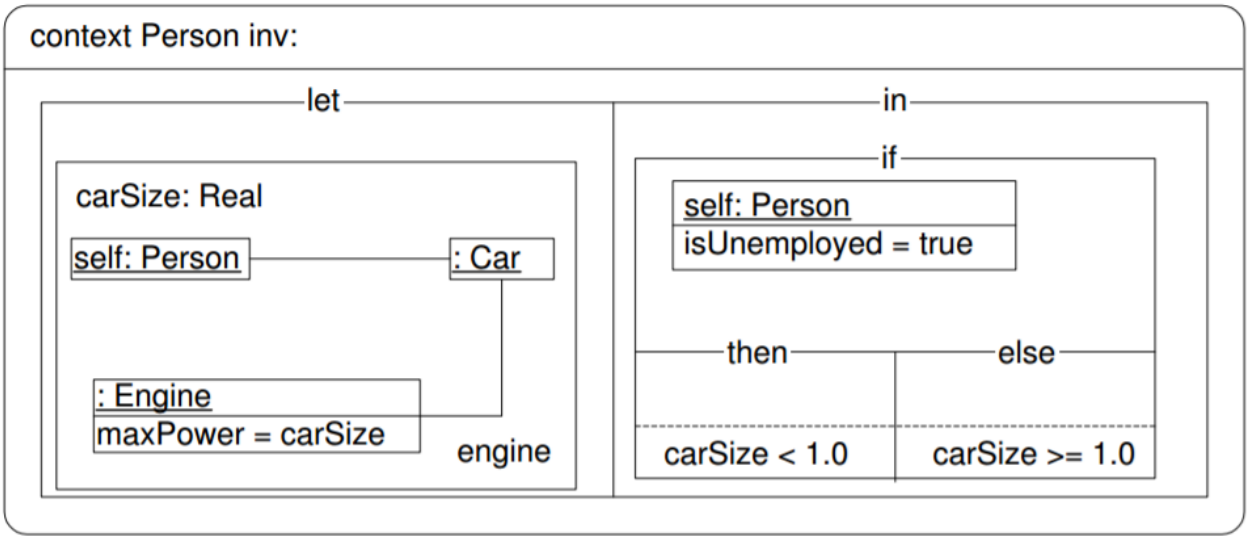

Рис. 1: Пример ограничения на VOCL.

2.5. Опыт QReal

В рамках проекта QReal кафедры системного программирования СПбГУ защищалась курсовая 2012 года, посвящённая визуальному языку заданию ограничений на модели [12]. Данная работа не была добавлена в QReal, но представляет собой ценность как первый опыт реализации языков ограничений в проектах кафедры. В данной курсовой работе был разработан формальный визуальный язык задания ограничений, создан соответствующий ему редактор, поддержан механизм проверки ограничений во время создания диаграмм на некотором визуальном языке, реализована выдача пользователю сообщений, в случае если некоторое ограничение не выполнено, реализован генератор ограничений, генерирующий по формальному описанию модели ограничений код в виде плагина, проверяющий эти ограничения, поддержана

возможность автоматически подгружать плагины ограничений после их генерации, не выходя из системы. Пример ограничения продемонстрирован на рис. 2.

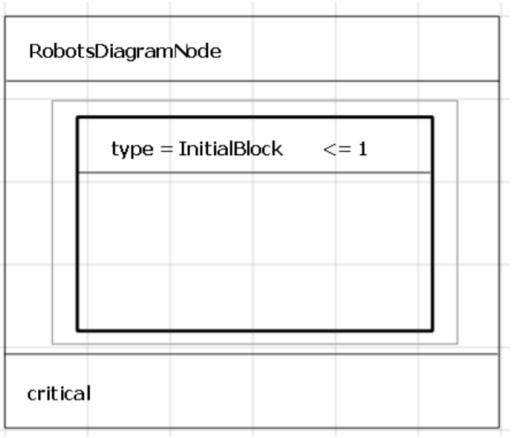

Рис. 2: Пример ограничения в QReal.

3. Реализация синтаксического анализатора языка OCL

3.1. Решение проблемы с неоднозначностью грамматики

Был заимствован подход Eclipse по решению проблемы с неоднозначностью грамматики. Он заключается в следующем: исходная грамматика расширяется путём объединения всех правил разрешения неоднозначностей, описанных в спецификации. В документации по OCL от Eclipse [3] подробно описывается получившаяся расширенная грамматика. Она была частично заимствована и изменена, в качестве изменения были декомпозированы большие синтаксические правила и проведены небольшие корректировки в соответствии со спецификацией OCL, на рис. 3 продемонстрирован фрагмент получившейся грамматики.

```
classifierContextDeclCS
:
        'context'
        (unrestrictedName ':')?
        nameExpCS
        (invCS | defCS)+
;
invCS
:
    'inv' unrestrictedName? ':' specificationCS
;
```

Рис. 3: Фрагмент грамматики.

3.2. Использование ANTLR-Tool

Также у Eclipse OCL была заимствована идея использования ANTLR-Tool [1] для генерации парсера. ANTLR реализует LL-анализ методом рекурсивного спуска [14]. ANTLR генерирует парсер по описанию EBNF-грамматики и создаёт дополнительные полезные файлы для постобработки построенного конкретного синтаксического дерева. В ANTLR Tool существует возможность генерации в несколько целевых языков: Java, C#, Python, Javascript, Go, C++, Swift.

Работу с ANTLR можно разбить на две фазы. В первой фазе (рис. 4) по входной грамматике генерируются вспомогательные файлы для парсера. Данное действие мы производим единожды, в дальнейшем просто используем в работе сгенерированные файлы.

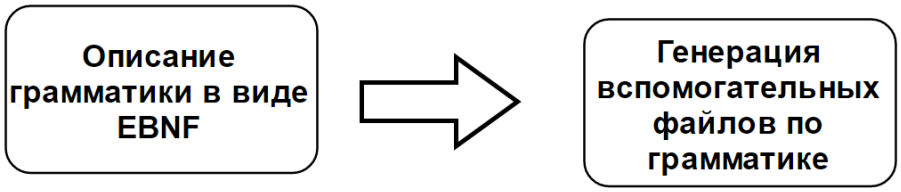

Рис. 4: Первая фаза работы с ANTLR.

Вторая фаза (рис. 5) относится конкретно к разбору выражений. Поступающее входное выражение с помощью средств ANTLR и сгенерированных по грамматике файлов преобразуется в конкретное синтаксическое дерево. Далее осуществляется постобработка построенного дерева с помощью средств ANTLR (Visitor или Listener) и получается абстрактное синтаксическое дерево. Существует два вида постобработки конкретного дерева в продукте ANTLR: через Listener или через Visitor. В первом подходе для обхода конкретного синтаксического дерева используется следующее: для каждого правила создаются методы enterRule и exitRule, вызывающиеся при входе в правило и выходе из него, соответственно. В этих методах описывается некоторая логика, позволяющая генерировать по конкретному синтаксическому дереву абстрактное или же производить специфические операции над ним. Во втором подходе, через Visitor, для каждого синтаксического

правила создаётся метод visitRule, в котором явно вызываются visit-методы потомков. Был выбран второй подход, поскольку работа через Visitor [13] нагляднее и код получается менее объёмным, нежели используя Listener.

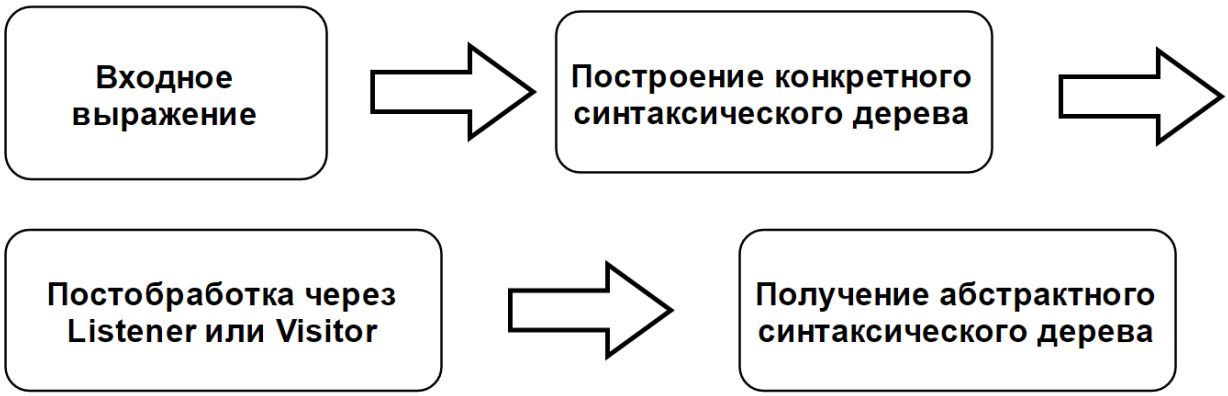

Рис. 5: Вторая фаза работы с ANTLR.

3.3. Архитектура

Парсер OCL реализован как отдельное решение, можно схематично разбить его на несколько компонент (см. рис. 6).

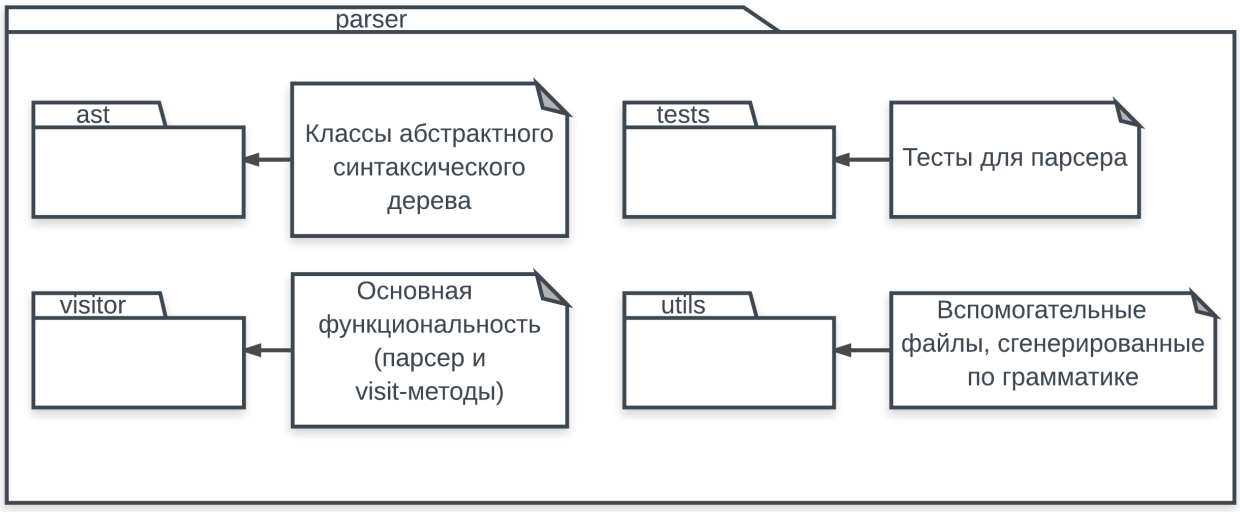

Рис. 6: Компоненты парсера.

На рис. 7 продемонстрирована архитектура компонента visitor, в зависимостях которого указаны ast и utils. Компонент ast используется в классе OCLVisitor при обходе конкретного синтаксического дерева для построения абстрактного дерева, классы которого и находятся в ast, utils используется для построения конкретного синтаксического дерева.

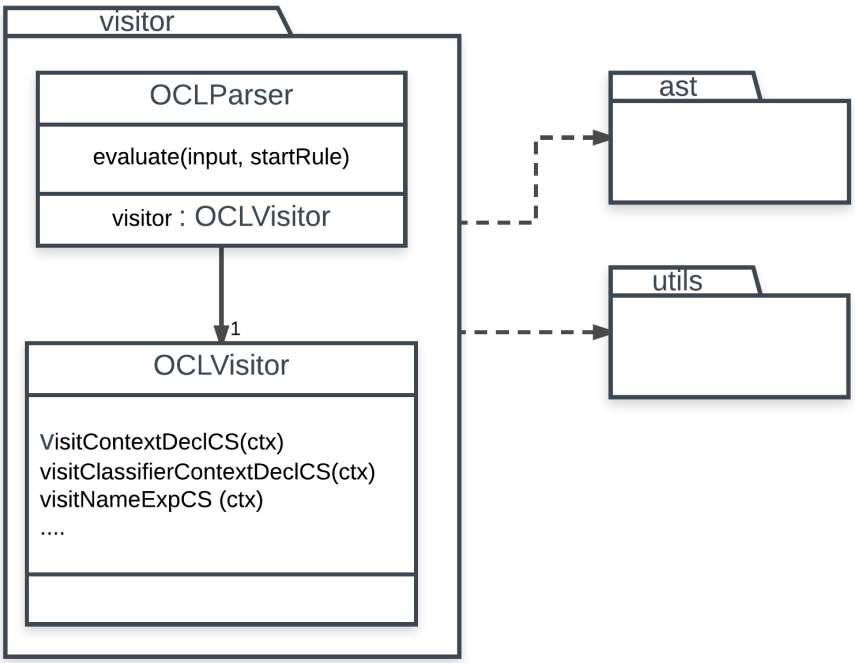

Рис. 7: Компонент visitor.

Рассмотрим компонент ast (рис. 8). Он состоит из трёх компонент: context decl expressions, core expressions и type expressions. Выражения из The Complete OCL, обеспечивающие передачу контекста, находятся в context decl expressions. Основные «строительные» блоки OCL находятся в core expressions. В type expressions соответственно находятся выражения, связанные с типами. Помимо этого, на рис. 8 изображён интерфейс ASTNode, каждый класс дерева реализует данный интерфейс.

Также отдельно можно рассмотреть классы, которые являются вспомогательными и не относятся к конкретным компонентам. На рис. 9 изображена архитектура классов, используемых для объявления переменных и операций.

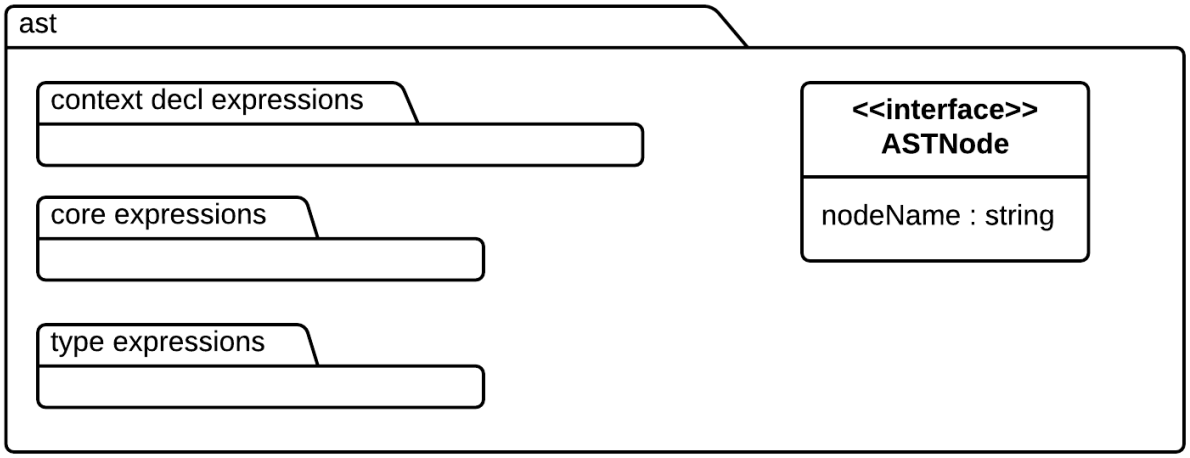

Рис. 8: Компонент ast.

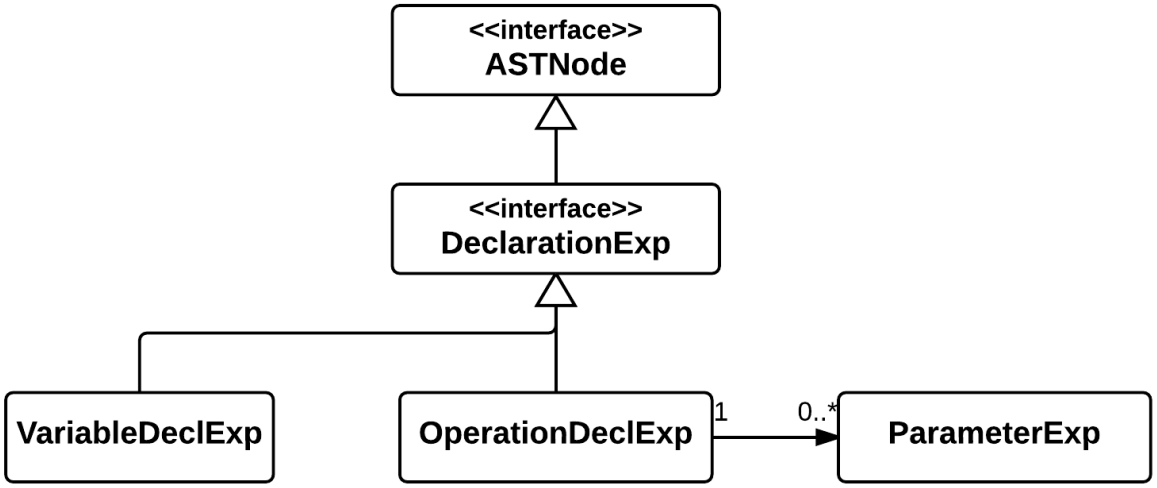

Рис. 9: Вспомогательные классы для объявлений переменных/операций.

Перейдём к рассмотрению context decl expressions (рис. 10). Класс ClassifierContextDeclExp отвечает за выражения, обеспечивающие указание контекста, для которого будет идти проверка ограничений.

Пример выражения:

```
context Company
inv: self.numberOfEmployees > 50
```

Класс OperationContextDeclExp отвечает за выражения, позволяющие накладывать ограничения на некоторые операции модели.

Пример выражения:

```
context Person::income(d : Date) : Integer
post: result = 5000
```

Класс PropertyContextDeclExp в свою очередь отвечает за выражения, позволяющие описывать ограничения на свойства модели.

Пример выражения:

```
context Person::income : Integer
init: parents.income->sum() / 100
```

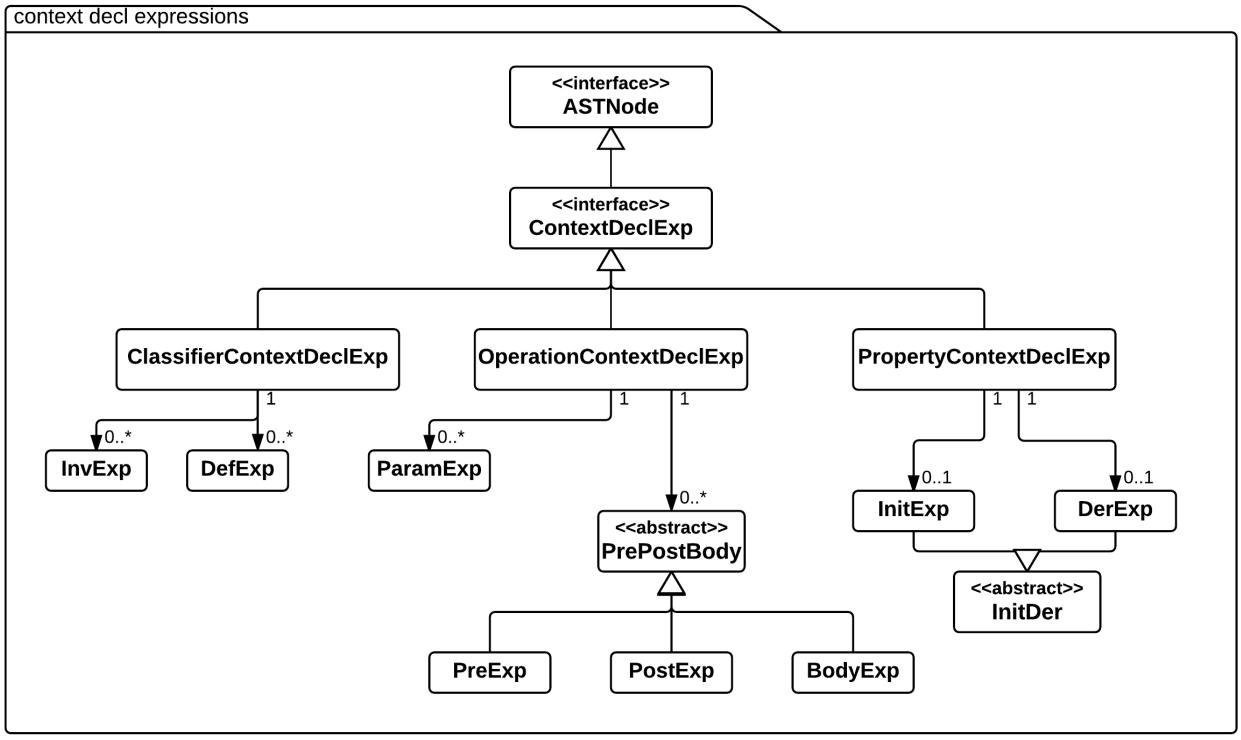

Рис. 10: context decl expressions.

Рассмотрим архитектуру core expressions (рис. 11).

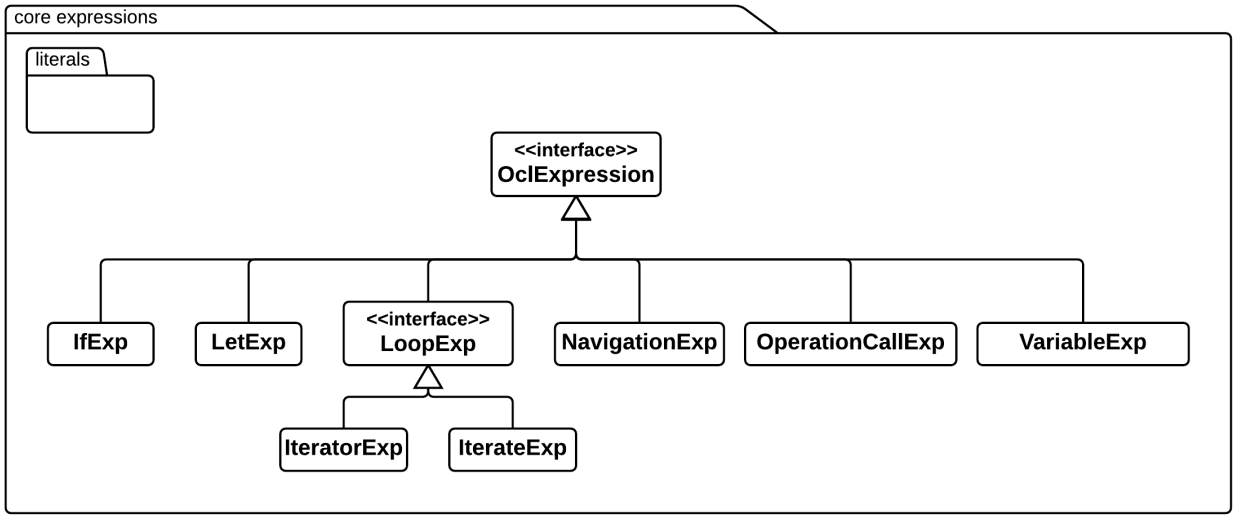

Рис. 11: core expressions.

Классы ifExp и letExp отвечают за условные выражения if-then-else-endif и объявления let, соответственно.

IteratorExp отвечает за итераторы OCL, описанные в спецификации.

Примеры итераторов:

```
context Company
inv: self.employee->forAll( p | p.age <= 65 )
inv: self.employee->forAll( e1, e2 : Person |
    e1 <> e2 implies e1.forename <> e2.forename)
inv: self.employee->exists( p | p.forename = 'Jack' )
```

IterateExp отвечает за выражения 'iterate'. Синтаксис:

```
collection->iterate( elem : Type; acc : Type = <expression> |
    expression-with-elem-and-acc )
```

Выражения 'iterate' имеют накапливающее значение (acc), которое обязательно должно быть инициализировано. Проходя по всем элементам коллекции, acc накапливает некоторое значение и возвращает его в конце в качестве результата. Это очень общая конструкция, с помощью которой можно реализовать многие итераторы.

NavigationExp отвечает за навигационные выражения, такие как "a.x", "a->x", "a.x()", "a->x()" и т.д.

OperationCallExp и VariableExp отвечают за представления операций и переменных, соответственно.

Важно отметить, что отсутствуют узлы, отвечающие за бинарные и унарные операторы, такие как ”+”, ”-” и т.д. Их отсутствие обусловлено тем, что в OCL выражения такие как, например, a + b эквивалентны a._’+’(b). Поэтому при разборе выражений, содержащих унарные и бинарные операторы, это учитывается и преобразуется в эквивалентную форму. То есть выражение a + b при разборе трансформируется в a._’+’(b), и результатом разбора будет NavigationExp, в котором источник навигации – a, цель навигации – операция ”+” с аргументом b.

Отдельно вынесен компонент literals, архитектура которого представлена на рис. 12.

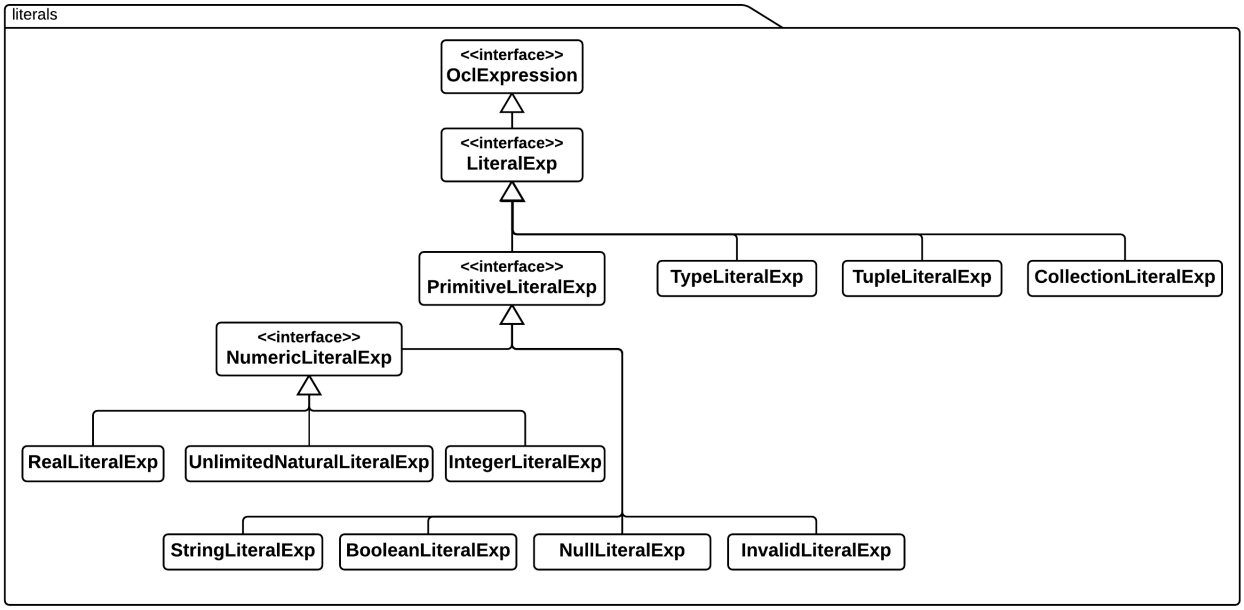

Рис. 12: literals.

Последним нерассмотренным компонентом ast является type expressions, архитектура которого представлена на рис. 13.

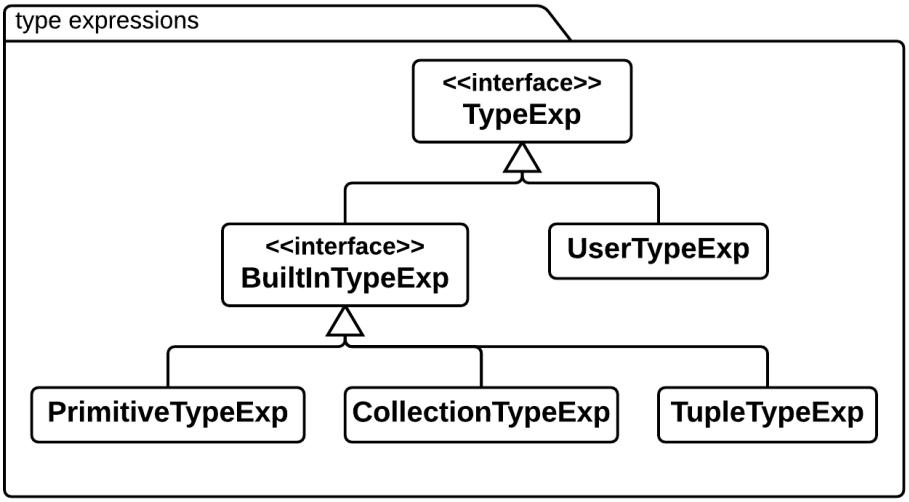

Рис. 13: type expressions.

4. Реализация интерпретатора языка OCL

4.1. Обзор поддержанных для интерпретации выражений

В данной работе были поддержаны выражения передачи контекста ClassifierContextDeclExp (см. рис. 10) из The Complete OCL. Данные выражения наиболее часто используемы, они позволяют задавать ограничения (выражения inv) и объявлять переменные/операции (выражения def) с областью видимости равной текущему контексту.

Пример выражения:

```
context A
def: x = 10
inv: self.y = x
```

Выражения из компонента core expressions были поддержаны полностью, а именно условные выражения, выражения let, операции, итераторы, выражения 'iterate' и т.д.

Также были поддержаны сокращения для навигаций. То есть при навигации от значения, не являющегося коллекцией, с помощью '->' происходит скрытое преобразование к коллекции Set, после чего интерпретируется навигация, будь то навигация к операции, итератору и т.д. Также при навигации от коллекции с помощью '.' происходит неявное преобразование к итератору collect, который был поддержан.

Пример скрытого преобразования:

```
aSet.name      преобразуется в      aSet->collect(name)
```

Рассмотрим итератор collect.

```
collection->collect(v : Type | expression-with-v)
collection->collect(v | expression-with-v)
collection->collect(expression)
```

Итератор collect пробегает коллекцию, возвращая результирующую коллекцию, содержащую вычисленные значения. При скрытом преобразовании мы получаем третью (сокращённую) форму collect. В тре-

тьей форме мы имеем доступ к полям объекта, являющегося элементом коллекции, напрямую. Следующие выражения эквивалентны:

```
self.employee->collect(person : Person | person.birthDate)
self.employee->collect(person | person.birthDate)
self.employee->collect(birthDate)
```

Важное замечание: при вызове на коллекциях типов Bag и Set результатом collect будет коллекция типа Bag, а при вызове на коллекциях типов Sequence и OrderedSet - типа Sequence.

4.2. Детали реализации

Интерпретатор, как и парсер, представляет собой отдельное решение, иными словами он не «зарыт» в коде проекта WMP, а является отдельной библиотекой, которую можно переиспользовать в любых проектах на языке Javascript. Основным классом, через который происходит общение с интерпретатором, является OCLInterpreter (рис. 14).

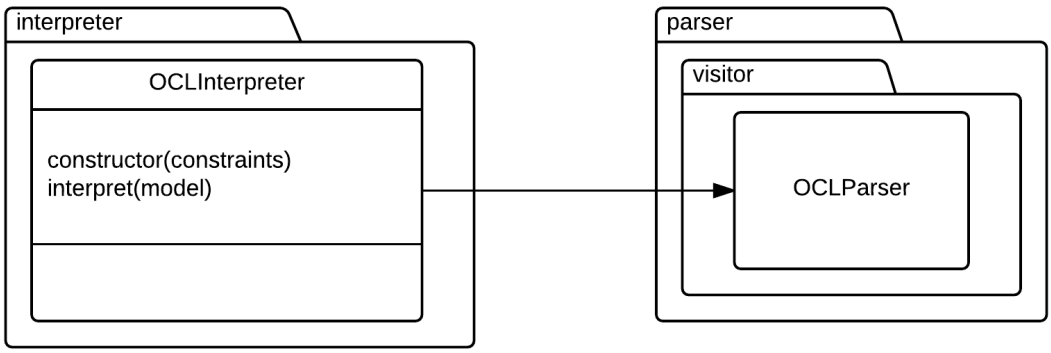

Рис. 14: OCLInterpreter.

В конструктор OCLInterpreter передаются OCL-выражения в виде строки, далее с помощью OCLParser строится абстрактное дерево для переданных выражений, парсер используется только в конструкторе, в дальнейшем используется построенное дерево. У OCLInterpreter есть единственный метод interpret(model), который принимает объекты модели, для которых будет идти проверка. Объекты модели, передаваемые в метод interpret, представляют собой Javascript-объект, у которо-

го ключи – это названия контекста, а значения – массивы экземпляров контекста с именем равным ключу.

Рассмотрим на следующем примере. Пусть у нас есть ограничение следующего вида:

```
context A
inv: self.str.substring(1, 3) = 'abc'
```

Допустим в нашей модели всего два экземпляра контекста 'A', для примера пусть она выглядит следующим образом:

```
let model = {
    A: [
        {
            str: 'abcdadkmv'
        },
        {
            str: 'abc1111'
        }
    ]
};
```

Для данного состояния модели ограничение выполнено.

В качестве результата интерпретации метод interpret возвращает объект, содержащий ошибки и предупреждения (errors и warnings) для каждого контекста. Для удобства рекомендуется указывать имена inv и def, которые являются опциональными, потому как эти имена учитываются в отчетах об ошибках. Ошибки могут возникать, например, если выражение inv после вычисления не равно true или имеет не булево значение. С предупреждениями немного сложнее, дело в том, что в OCL при вычислении выражений, в отличии от других языков, многие некорректные конструкции не прекращают проверку ограничений, а возвращают в качестве вычисленного значения ошибочной конструкции значение invalid. Достаточно сложить строку с числом, либо вызвать несуществующую операцию для того, чтобы получить invalid. С точки зрения OCL интерпретация выражения продолжится, но полезно получать предупреждения в таких ситуациях, иначе будет непонятно,

из-за чего ограничение нарушено.

4.2.1. Модель

Тут хотелось бы подробно осветить представление модели, для которой осуществляется проверка ограничений. В примере выше можно увидеть, что экземпляр контекста является обычным Javascript-объектом. На самом деле интерпретатор довольно универсален и экземпляром контекста может являться любое значение (число, булево значение, строка, объект, массив – частный случай объекта, null, undefined).

Допустим экземляр контекста – число, рассмотрим возможное ограничение на такую модель:

```
    context A
    inv: self.abs() = 10
```

И сама модель, допустим, представляет собой следующее:

```
let model = {
    A: [10, -10]
};
```

Ограничение выполнено, но, как правило, под экземпляром контекста понимается именно Javascript-объект, остановим свой взгляд именно на таком случае. Рассмотрим на примере некоторой модели нюансы интерпретации, для простоты положим, что у нас всего один экземпляр контекста 'A', и модель выглядит следующим образом:

```
let model = {
    A: [
        {
            x: 10,
            y: {
                z: 21
            },
            elements: [1, 2, 3, 4],
            self: 54,
            u1: null,
            u2: undefined,
            elems: [{x: 101}, {x: 100}]
```

```
        }
    ]
};
```

Решение представления экземпляра контекста в виде Javascript-объекта обусловлено тем, что это очень удобно в использовании, потому как нет необходимости приводить представление модели к какой-либо специфической форме. Рассмотрим некоторые возможные ограничения на данную модель:

```
context A
inv: self.x = 10
inv: self.y.z = 21
inv: self.elements->asSequence()->at(1) = 1
```

Данные ограничения весьма просты, но показательны. В первых двух ограничениях демонстрируется навигация по объекту. В третьем ограничении мы обращаемся к массиву elements, хранящемуся в экземпляре контекста, далее преобразуем его к коллекции типа Sequence и достаем первый элемент. Отметим, что в OCL, в отличии от многих языков программирования, индексы начинаются не с 0, а с 1. Очень важно заметить, что при навигации, если мы получаем массив, то преобразуем его к коллекции вида Bag. Так в третьем ограничении self.elements автоматически преобразуется к коллекции Bag{1, 2, 3, 4}. Мы осуществляем преобразование к Sequence, потому как у Bag нет операции at(index).

Напомним, что при сравнении неупорядоченных коллекций порядок элементов не учитывается и наоборот, то есть:

```
context A
 inv: self.elements = Bag{1..4} //true
 inv: self.elements = Bag{4, 2, 3, 1} //true
 inv: self.elements->asSequence() = Sequence{1..4} //true
 inv: self.elements->asSequence() = Sequence{4, 2, 3, 1} //false
```

Сравнение объектов осуществляется по ссылке.

Рассмотрим ещё пример:

```
context A
 inv: self.elems.x = Bag{101, 100} //true
 inv: self.elems->collect(elem | elem.x) = Bag{101, 100} //true
 inv: self.x->asSequence()->at(1) = 10 //true
 inv: null->isEmpty() //true
 inv: self.self = 54 //true
 inv: self.u1.oclIsUndefined() and self.u2.oclIsUndefined() //
   true
```

Первый пример демонстрирует скрытое преобразование коллекции и эквивалентен второму примеру. Третий пример показывает скрытое преобразования к коллекции, self.x = 10, далее преобразовывается к Set{10}, далее вызовом операции asSequence() к Sequence и далее первый, он же единственный, элемент сравнивается с 10. В четвёртом примере демонстрируется особенность значения null, согласно спецификации при преобразовании null к коллекции Set получаем пустую коллекцию. При преобразовании invalid к Set, например, получаем invalid, все подобные особенности описаны в спецификации [5]. В предпоследнем примере показана корректность работы в случае, когда поле объекта имеет имя 'self'. Последний и очень важный пример показывает следующее: при обработке Javascript-значений null и undefined, они вычисляются как null. Но есть один интересный момент, в Javascript при обращению к полю, которого нет у объекта, возвращается undefined. В данной реализации это учитывается, и если поле у объекта есть и равно null или undefined, то интерпретируется как OCL-значение null, иначе, если данного поля нет, возвращается invalid, и в warnings записывается предупреждение о попытке вызова несуществующего свойства на объекте.

Неосвещённой осталась конструкция def, позволяющая объявить переменную/операцию для использования в выражениях inv.

Рассмотрим пример:

```
context A
 def: x = 10
 def: fact(n:var) = if n = 1 then n else n * fact(n - 1) endif
 inv: self.x = x //true
 inv: fact(5) = 120 //true
```

Данный пример демонстрирует способ объявления переменной, а также операции. Более того, объявленная функция fact(n) – рекурсивная. Стоит также отметить, что нельзя объявить переменную с именем равным 'self' или псеводонимом для 'self'.

4.3. Поддержка OCL Standard Library

В данном разделе перечислены реализованные операции из OCL Standard Library. За подробным описанием операций следует обращаться к спецификации [5].

OclAny: =, <>, oclIsUndefined, oclIsInvalid, oclAsSet.
OclVoid: oclAsSet.
OclInvalid: =, <>, oclAsSet.
Операции с числами: +, –, *, /, <, <=, >, >=, abs, max, min, toString.
String: +, <, <=, >, >=, size, substring, toUpperCase, toLowerCase, indexOf.
Boolean: or, xor, and, not, implies, toString.
Collection: oclAsSet, asSet, asOrderedSet, asSequence, asBag, size, isEmpty, <>, includes.
Bag: asBag, =.
Set: oclAsSet, asSet, =.
OrderedSet: asOrderedSet, =, at.
Sequence: asSequence, =, at.

Тип OclVoid представляет значение null, OclInvalid – invalid. OclAny – базовый тип для всех остальных, его операции наследуются всеми, некоторые их переопределяют. Также все коллекции наследуют операции Collection. Коллекции переопределяют методы преобразования к

их типу, например Bag переопределяет метод asBag(), это обусловлено тем, что при подобных преобразованиях ничего не происходит, возвращается исходная коллекция.

Также поддержаны итераторы forAll и collect.

5. Внедрение в проект WMP

Как было замечено ранее, парсер и интерпретатор реализованы как отдельные решения. Для их использования в языке JavaScript достаточно взять проект с репозитория [1], либо при использовании менеджера пакетов npm [8] достаточно в package.json проекта добавить в зависимости данный репозиторий.

Для апробации в проекте WMP было решено протестировать набор ограничений на связи между блоками на диаграмме. Проверялись следующие ограничения: у каждой связи есть источник, у каждой связи есть цель, у каждой связи есть логический id, также проверялось, что значения свойства Guard у каждой связи принимает значение по умолчанию ('true', 'false', 'iteration'), либо пусто. Описание ограничений хранится в отдельном файле, пример данного файла для нашего набора ограничений продемонстрирован ниже:

```
- -Invariants for all links
context Link

inv hasSource: not self.jointObject.attributes.source.id.
   oclIsUndefined()

inv hasTarget: not self.jointObject.attributes.target.id.
   oclIsUndefined()

def: guardValue = self.changeableProperties.Guard.value
inv defaultGuardValue: Bag{'false', 'true', 'iteration', ''}->
   includes(guardValue)

inv hasLogicalId: not self.logicalId.oclIsUndefined()
```

Был создан класс ConstraintsChecker, в конструкторе которого с помощью реализованного парсера OCL происходит разбор OCL-выражений, содержащихся в файле с ограничениями. Данный класс обладает двумя методами: «обновить модель» и «проверить ограничения на вы-

[1] https://github.com/DenisKogutich/ocl

полнимость». Второй метод использует реализованный интерпретатор OCL-выражений. Далее, в классе SceneController, отвечающем за сцену в проекте WMP, происходит создание экземпляра ConstraintsChecker. При добавлении связи, удалении связи, изменении свойства и изменения положения связи происходит обновление модели и проверка ограничений на выполнимость.

Благодаря тому, что в данной реализации экземплярами контекста являются JavaScript-объекты, не пришлось осуществлять какие-либо преобразования для того, чтобы привести состояние модели к приемлимой форме. Для осуществления проверки данных ограничений объекты, отвечающие за представления связей в проекте WMP, были переданы «как есть». Хотя, конечно же, при более сложных наборах ограничений присутствует вероятность необходимости приведения модели к какому-либо представлению, отличающемуся от используемого «как есть» в проекте.

Заключение

В рамках данной работы была реализована возможность задания ограничений в проекте WMP. Были выполнены следующие задачи:

- создан парсер таких подножеств языка OCL как The Essential OCL и The Complete OCL;
- создан интерпретатор OCL-выражений, для интерпретации поддержаны выражения из The Essential OCL и выражения объявления контекста ClassifierContextDecl из The Complete OCL, включающие в себя конструкции inv и def, также частично поддержаны операции и итераторы из OCL Standard Library (подробнее см. раздел 4.3);
- реализованные парсер и интерпретатор тщательно протестированы;
- реализованные парсер и интерпретатор апробированы в проекте WMP.

Исходные коды опубликованы в репозитории проекта на GitHub[2].

[2] `https://github.com/DenisKogutich/ocl`

Список литературы

[1] ANTLR. — URL: www.antlr.org/ (online; accessed: 12.04.2017).

[2] Andrew Fish John Howse Gabriele Taentzer, Winkelmann Jessica. Two Visualizations of OCL: A Comparison. — URL: http://www.mathematik.uni-marburg.de/~swt/Publikationen_Taentzer/VOCLTR.pdf (online; accessed: 14.05.2017).

[3] OCL Documentation. — 2014. — URL: download.eclipse.org/ocl/doc/5.0.0/ocl.pdf (online; accessed: 12.04.2017).

[4] OCL – Eclipsepedia. — URL: wiki.eclipse.org/OCL (online; accessed: 12.04.2017).

[5] OCL.book. — 2014. — URL: www.omg.org/spec/OCL/2.4/PDF/ (online; accessed: 12.04.2017).

[6] OCL.js. — URL: ocl.stekoe.de/ (online; accessed: 04.05.2017).

[7] The Modelling Simulation, lab (MSDL) Design. Object Constraint Language. — URL: http://msdl.cs.mcgill.ca/presentations/02.06.07.OCL/presentation.html (online; accessed: 14.05.2017).

[8] npm. — URL: https://www.npmjs.com/ (online; accessed: 14.05.2017).

[9] qreal/wmp Wiki. — URL: https://github.com/qreal/wmp/wiki (online; accessed: 14.05.2017).

[10] «ИНТУИТ» Национальный Открытый Университет. Ограничения целостности и язык OCL. — URL: http://www.intuit.ru/studies/professional_skill_improvements/1426/courses/74/lecture/27919?page=4 (online; accessed: 14.05.2017).

[11] А.В. Безгузиков. Микросервисная архитектура QReal-Web // Курсовая работа. — 2016. — URL: http://se.math.spbu.ru/SE/

YearlyProjects/spring-2016/344/344-Bezguzikov-report.pdf (online; accessed: 12.04.2017).

[12] А.О. Дерипаска. Визуальный язык задания ограничений на модели в QReal // Курсовая работа. — 2012. — URL: se.math.spbu.ru/SE/YearlyProjects/2012/YearlyProjects/2012/345/345_Deripaska_report.pdf (online; accessed: 12.04.2017).

[13] Влиссидес Э. Гамма Р. Хелм Р. Джонсон Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. — Питер, 2016.

[14] Мартыненко Б.К. Языки и трансляции. — Издательство С.-Петербургского Университета., 2004.